Report No. WH97JR00-A002

REAL-TIME NETWORK MANAGEMENT

FINAL TECHNICAL REPORT

July 1998

Sponsored by

Defense Advanced Research Projects Agency
ITO

Issued by U.S. Army Aviation and Missile Command Under

Contract No. DAAH01-98-C-R040

DISTRIBUTION STATEMENT
Approved for public release; distribution unlimited.

Report No. WH97JR00-A002

REAL-TIME NETWORK MANAGEMENT

Synectics Corporation
111 East Chestnut Street
Rome, New York 13440

Joseph R. Riolo, Principal Investigator

Telephone: 315-337-3510

Effective Date of Contract: 24 November 1997
Contract Expiration Date: 24 July 1998

RTNM Final Technical Report

Table of Contents

Table of Figures

1.0 INTRODUCTION

This document is the Final Technical Report (Contract Data Requirements List [CDRL] A002) for contract number DAAH01-98-C-R040 entitled, "Real Time Network Management." This was a 7-month effort, which ran from 24 November 1997 to 24 July 1998. Synectics was the prime contractor with the State University of New York (SUNY) Institute of Technology as a subcontractor on the effort.

Synectics Corporation submitted a proposal abstract, in response to DARPA SBIR SB972-076 (Real-Time Network Performance Diagnosis), for developing methods for diagnosing network performance problems in real time. It supported the dominant theme of the Phase I SBIR with research into the problem and development of a prototype.

The first phase of the project was to develop a prototype system that included interactive graphics for describing the components of a local area network (LAN) and its interconnections to other LANs or internetworks. There were three major considerations for this development effort.

- The description had to include hooks for modeling the expected behavior (bandwidth, latency, queuing, etc.), and for monitoring the runtime behavior via traffic probes, Simple Network Management Protocol (SNMP), remote monitoring, load measurements, etc.
- The modeled and observed behaviors had tot be displayed in a lucid manner that assisted analysts and operators, and could be easily modified.
- The LAN descriptions must be modifiable and composable in a simple fashion.

2.0 OBJECTIVES

The objective of our effort was to develop methods for diagnosing network performance problems in real time for DARPA. According to our Phase I research, it is possible to collect data on the network and morph it into queuing models to produce information about the network and physical layers of nodes on a network.

The program consisted of three tasks. In the first task Synectics assembled a set of metrics that formed an accurate network traffic model, identified mechanisms for hooking well-researched mathematical performance models into software artifacts to allow the definition of expected behavior, and identified deviations from such behavior. Secondly, Synectics described the logical architecture of the proposed performance monitoring system as an object model. Finally, Synectics developed the prototype using Java, with support from JMAPI (Java Management

Application Programmer's Interface) and JDBC (Java Database Connectivity), and the object model designed in the second task.

2.1 TASK 1 - MATHEMATICAL MODELING

A networking environment is a dynamic entity encompassing a finite number of objects, each functioning to operate within a specific set of constraints. Our goal was essentially to monitor this environment, measure and make inferences on its observables, derive the appropriate states of interest, and report them appropriately in order to provide a diagnostic basis to network managers to draw their attention to exceptional and abnormal events. Indeed, to appropriately accomplish these activities and reasonably correctly in a real-time framework, we needed various models of our object to depict both its own state and that of its constituents. A model of the entire object as a single entity may not be suitable at its constituent levels. Not only should we be able to appropriately model each one of the network objects, such as a switch or a link, but we should also be able to model the entire network as a single entity given our models of its constituents. Secondly, a single model of a dynamic entity may not be adequate. Even if such a model depicts only stationary states, as they almost always do, one may be forced to consider different types of models to handle different equilibrium situations. For instance, at a low traffic load, we might use one type of model for an entity, but when the traffic load becomes heavy, we might have to use another model to more accurately depict the activity there. If this is desirable, then switching from one model to another must be triggered by the presence of an event at a meta-level of the models.

The local states as perceived at each object site are derived from the raw observed data (the observables) as appropriate aggregates. The global system state is a list of all aggregate local states of the network along with an appropriate set of computed temporal invariants based on inter-object coupling over the local state space.

The object network is seen as a hierarchical abstraction as indicated below.

NETWORK = {Nodes, Links}
Nodes = {Ports, Switches}
Link = (Physical connection between two nodes) . . . (1.0)

A local state variable $\vartheta_{obj}(t)$ is an attribute of an object at a time t, and we assume that its short-term time-averaged value is "stable" and is denoted by $\overline{\vartheta}_{obj}^{epoch}(t)$ for a specific epoch. The long-term time averaged value of this state is the quantity (which converges to a steady-state value if the network converges to a steady state).

$$\bar{\vartheta}_{obj} = \frac{1}{n_{ep}} \int \bar{\vartheta}_{obj}^{epoch} d(epoch) \qquad \ldots (1.1)$$

where n_{ep} is the epoch density over the period of observation on which the long-term behavior is sought. Our reporting profile is envisaged as follows. For every object *obj* at any arbitrary time *t,* the object manager is ready to report the following.

obj = (object ID, object class)
time = current_time t

$$state\_list(t) = \vartheta(t) = (\vartheta_1(t), \vartheta_2(t), \ldots, \vartheta_p(t)) \quad \ldots (1.2a)$$

$$previous\_state\_aggregate(epoch\_number = -1) = \bar{\vartheta}^{ep=-1} = (\bar{\vartheta}_1^{-1}, \bar{\vartheta}_2^{-1}, \ldots \bar{\vartheta}_p^{-1}) \quad \ldots (1.2b)$$

over current and previous epochs ep_0 and ep_{-1}, respectively. Note that $state\_list(t)$ is always the current state of the object in the current epoch ep_0.

The *estimated state* list at the next epoch ep_{+1}, as inferred from the current and the past behavior through an appropriate intelligent estimator (we assume that we have appropriate model(s) necessary to compute this), look like

$$next\_state\_aggregate(epoch\_number = +1) = \hat{\vartheta}^{+1} = (\hat{\vartheta}_1^{+1}, \hat{\vartheta}_2^{+1}, \ldots, \hat{\vartheta}_p^{+1}) \qquad \ldots (1.2c)$$

And finally, the object would post its long term average of the state list over all the epochs in a given time-interval *T* as

$$\bar{\vartheta}_T = (\bar{\vartheta}_1, \bar{\vartheta}_2, \ldots, \bar{\vartheta}_p, \ldots, \bar{\varsigma}_{1G}, \bar{\varsigma}_{2G}, \ldots, \bar{\varsigma}_{qG}) \quad \ldots (1.2d)$$

In this case, a state variable $\bar{\vartheta}_M$ is the long-term average list of the state of the object *M,* while the $\bar{\varsigma}_{mG}$ is the long-term average of some computed variable

$$\varsigma_{mG} = f_G(\vartheta_1, \vartheta_2, \ldots, \vartheta_p) \qquad \ldots (1.2e)$$

which has a global relevance and a semantic such as congestion level. We assume that, on whatever model we propose to project this measure, it is operationally stable and observable over time and it consists of two spatially separated objects P and Q.

$$\varsigma_{mG}(t + \delta t) \approx \varsigma_{mG}(t) \text{ and}$$
$$\varsigma_{mG}(P) = \varsigma_{mG}(Q) \qquad \ldots (1.2f)$$

Note that all these states were either derived from the local observables and/or inferred from both the previous and the current states. The local observable at an object switch may be minimally the number of messages and requests queued and being serviced there, the server service rate, the arrival rate at the server queues, or the service discipline (e.g., FIFO [first in, first out], priority,

polling, etc.). If it were a link, the local observable could be its capacity (static) and its data flow-rate (dynamic). For a switch, it could also include the fraction of packets (or cells) dropped, packet injection rates at the links to which it is connected, or the input/output queue densities at the buffers to the links.

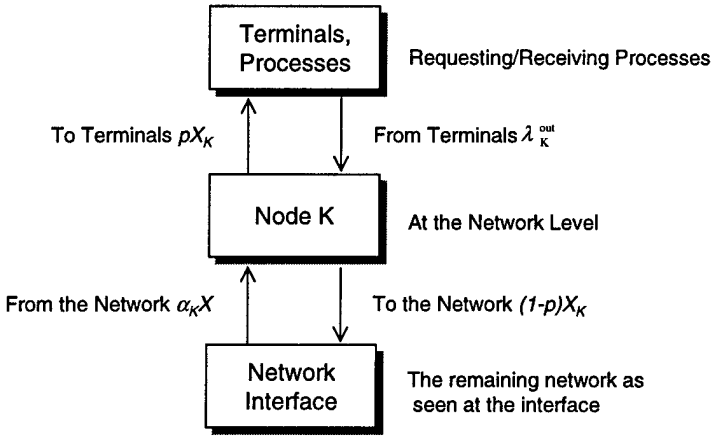
2.1.1 LOCAL PERSPECTIVE

We define *local perspective* as a view of a single network object in the context of the entire network in which it is embedded. In this section we will focus on the state computation of a single node or a link in a network. We will offer a low-to-moderate traffic description of a server node in an M/M/1/k architecture. Other realistic models would be delivered in subsequent reports.

2.1.1.1 M/M/1/K MODEL OF A SINGLE SERVER

The environment seen from a node appears as follows at the network layer level.

Figure 1. Network Layer Level



The packet throughput at a node K, X_K, is apportioned between two streams. The inbound stream moving toward the connecting stations and processes receives pX_K of the node traffic. The remaining traffic appears at the interface to be injected into the network as seen at the interface. If the loss at the interface is $loss_{int}$, then the amount going into the network is $(1-loss_{int})(1-pX_K)$. Similarly, if a portion of the incoming traffic from Transmission Control Protocol (TCP) and Internet Control Message Protocol (ICMP) layers is dropped at the node K, the actual traffic entering from above into the node λ_K^{out} would be less than what these nodes are sending to the node K.

For our convenience, let us assume that the Jackson assumption holds and the node is seen as a finite buffer Markovian server so that it could be modeled as an *M/M/1/k* system (k being the maximum number of packets the buffer is allowed to hold). In this perspective,

The average total traffic entering into the node, $\lambda_K = \lambda_K^{out} + \alpha_K X$ pkts/sec
The average packet transmission rate at node K = μ_K pkts/sec
The buffer size at the node = k

The blocking probability at the node $p_K^{block} = \frac{1-\rho}{1-\rho^{k+1}}\rho^k$ given that its total buffer volume is for k packets. Here $\rho = \frac{\lambda_K}{\mu_K}$. We also obtain the following equilibrium results.

(a) The average number of packets in the node (at the queue and at the server)
$N_K = \frac{\rho}{1-\rho} - \frac{(k+1)\rho^{k+1}}{1-\rho^{k+1}}$ if $\lambda_K \neq \mu_K = \frac{k}{2}$, otherwise

(b) The effective average arrival rate into the node is $\lambda_K(1-p_k)$

(c) Using, Little's law, the expected response time at the node $T_K = \frac{N_K}{\lambda_K(1-p_k)}$

(d) Average node utilization rate is $\rho(1-p_k)$

(e) The node idle probability $p_0 = \frac{1-\rho}{1-\rho^{k+1}}$

(f) Queuing time spent in the buffer $T_K^{que} = \frac{N_K - 1 + p_0}{\lambda(1-p_k)}$

For convenience, we would use the notation $A6$ to denote an appropriate time average (or a weighted average) of an observable $\theta(t)$ over an observation period of size T. We assume δt to be the size of a monitoring interval.

$$A\theta = \frac{1}{T}\int \delta(\theta_{t+\delta t} - \theta_t)dt$$

Then, in terms of our observables as obtained from the Management Information Base (MIB), the following would be realized for a node.

(a) The average packet traffic flow going into the network, $(1-p)X_K = \frac{A(if17 + if18)}{\delta t}$

(b) The average traffic load delivered to the network layer $\alpha_K X = f((A if12 + \Delta if11)/\delta t)$ where $f(.)$ has to be determined.

(c) The average traffic load delivered to the upper-layer from the node $pX_K = \frac{A ip3}{\delta t}$

(d) The average traffic load delivered from the TCP/ICMP to the internet protocol (IP) layer of the node $\lambda_K^{out} = \frac{\Delta ip\ 10}{\delta t}$

(e) The blocking probability at the IP level $p_k = \frac{\Delta ip\ 11}{\Delta ip\ 10} = \frac{1-\rho}{1-\rho^{k+1}}\rho^k$

2.1.1.2 QUEUING MODELS FOR SWITCHES (ROUTERS), NODES, AND PORTS

We assume that, for our switches, nodes, and ports, these could be described by appropriate queuing models M/M/1, M/M/1/k, M/G/1, etc., if the server were a single server or could be modeled as a single server.

A typical node is modeled as a tandem queue. We assumed Jackson's assumption holds in the sense that the queues are separable even though individually they might not be M/M/1 queues. We considered a node (or a port) as a tandem queue served by two sets of servers, the IP-SAP (Service Access Point) and the server at the interface, which we simply called the interface server. The interface server transmits requested frame-traffic to the network and receives frames to send up via the data-link layer to the IP-SAP, or the IP server. We assumed the service rates at the IP layer and at the interface are given by the parameters μ_1 and μ_2, respectively. For convenience, we let $X_1 = \lambda_1$ and $X_2 = \lambda_2$. Given these, we defined traffic densities at the servers as $\rho_1 = \lambda_1 / \mu_1$ and $\rho_2 = \lambda_2 / \mu_2$. In terms of these variables the node's states could be obtained using the appropriate queuing model.

One major problem here was the estimation of the IP server rate μ_1. There is no appropriate SNMP variable by which to obtain this information. Yet, at such a service site where packet fragments are to be reassembled, the IP-SAP has to wait for those extreme cases where the fragments do not turn up until the last moment. (Some never arrived.) This was a time-intensive operation and therefore could not be ignored. One could approximate the average, effective packet service time as

$\frac{1}{\mu_1} \approx \frac{U\delta t}{(n^s_{reassemb} + n^f_{reassemb})}$, where $n^{s,f}_{reassemb}$ denotes the number of successful and unsuccessful packet reassembly operations within the given time interval δt. It is reasonable to assume that $n^f_{reassemb}$ is most likely positively correlated with the total number of fragments received at the IP layer which, in turn, is linearly dependent on the incoming traffic rate λ_1. The factor U appears here as the server's utilization factor. We could express this as the requirement that

$n^f_{reassemb} = \eta\, n^s_{reassemb}$, where $\eta = \frac{total\ fragment\ received}{total\ correct\ fragments} - 1$. Then our effective service time becomes

$$\frac{1}{\mu_1} = \frac{U\delta t}{n^s_{reassemb}(1+\eta)}$$

For each node, we would obtain from the SNMP data:

(a) The average packet arrival rate into the system λ_1 pkts/sec at the IP layer, and the λ_2 frames per sec at the interface.

(b) The average service time for a packet $\frac{1}{\mu_1}$ sec at the IP layer and $\frac{1}{\mu_2}$ sec at the interface.

(c) The average number of packets dropped at the node per unit time n_d^I pkts where the index $I \in \{Ip\_layer, Interface\}$.

(d) The variance of the inter-arrival times $\sigma^2_{I/\lambda}$ obtained from the past data at both the IP layer and the interface.

(e) The variance of the service time $\sigma^2_{I/\mu}$ obtained from the past data at both the IP layer and the interface.

If $n_d = 0$, then we are effectively dealing with an infinite buffer queuing system (buffer-size k $\rightarrow \infty$). If the variance $\sigma^2_{I/\mu} \approx \frac{1}{\mu^2}$, then we contend that the server is exponentially distributed. Normally, we would assume the arrival process to be Poisson distributed in which case the inter-arrival time must be exponentially distributed with a variance $\sigma^2_{I/\lambda} = \frac{1}{\lambda^2}$. However, if we find the variance substantially low, say, $1/q\lambda^2$, where q is greater than 2, then we consider the arrival process to be Erlang-q distribution. At an interface, the service time is essentially the transmission time of a frame.

2.1.1.3 MODEL SELECTION RULES

(a) The default model would be M/M/1 for each queuing subsystem.

(b) The current model would be the last selected model unless it is switched to another one.

(c) If $n_d = 0$, then the selected model would be one of M/M/1, M/G/1, M/E_k/1, E_k/M/1, M/D/1, or GI/G/1.

(d) If $n_d \neq 0$, then the switched model would be M/M/1/k. The blocking probability p_k for the model would be computed as $p_k = n_d / \lambda$.

(e) Once the underlying model for a node changes to a new one, the earlier derived state results would not be used to infer the future results. Note that this is valid only for "derived" results. The observed or monitored variables would still be inferred when needed from past results.

(f) If the variance of the arrival distribution is about $\frac{1}{m\lambda^2}$ for an arrival marked with λ, then the selected model would be E_m for the arrival part. Similarly, if the variance of the service distribution is about $\frac{1}{n\mu^2}$ for the service marked with μ, then the selected model would be E_n for the service side. Thus, a chosen model might appear as $E_m/E_n/1$ instead of M/G/1 if both the arrival and the service time distributions appear as Erlangan.

(g) If the length of the service time per entity is not exponentially distributed (i.e., service rate variance is different from $1/\mu^2$ when the expected service time is $1/\mu$), we would use a general service time distribution and switch to an M/G/1 or GI/G/1 model. A GI arrival distribution would be considered if the inter-arrival times are independent and identically distributed. We would use the GI/G/1 type model, particularly in a heavy traffic load situation when traffic intensity ρ is near to 1.0.

(h) If the service time per packet/frame is effectively a constant, then we would use the M/D/1 model. This would be the preferred model for ports in a bridge.

2.1.1.4 M/M/1 MODEL

The parameters for this model are:

- Arrival rate λ pkts/sec into a queue , and
- Service rate μ pkts/sec of the server.

These are time averaged over an interval during which the model is assumed to be valid. Given these, we would then compute the following.

(a) The traffic intensity, $\rho = \frac{\lambda}{\mu}$ Erlang

(b) The server utilization, $U_{node} = \rho$

(c) The probability that the number of packets or frames in the system is no less than n, $p(N \geq n) = \rho^n$

(d) The average number of packets or frames in the system $L = \frac{\rho}{1-\rho}$

(e) The average number of packets or frames in the queue $L_q = \frac{\rho^2}{1-\rho}$

(f) The residence time (the waiting time) in the system $T = \frac{1}{\mu(1-\rho)}$

(g) The average queuing time in the server queue $T_q = \frac{\rho}{\mu(1-\rho)}$

(h) The queuing time that r percent of the customers do not exceed, i.e., the rth percentile queuing time $\pi_q(r) = T\,ln\left(\frac{100\rho}{100-r}\right)$

(i) The rth percentile residence time $\pi_T(r) = T\,ln\left(\frac{100}{100-r}\right)$

If the average service time is not directly available, one could compute it as a derived variable $\mu_{effective} = T + \lambda$ and $\mu_{effective} = \frac{2\lambda}{-L_q + \sqrt{L_q^2 + 4L_q}}$, provided $L_q \neq 0$. This may be necessary when the server is perceived as a logical server such as at a higher level in a protocol suite.

It is possible that even though the computed ρ might appear to be greater than zero, the observed queue length at a node may be zero at a specific time point. Note that these are all time-averaged measures; they need not always correspond to specific observation at a given time point. Also, all percentile formulations would yield negative values when ρ is small; in such events, all negative values should be reported as zero.

2.1.1.5 M/M/1/K MODEL

In this case, the server has a finite buffer of size k to accommodate incoming packets/frames. This would be indicated when we observe packets/frames that we assume are discarded owing to lack of buffer. We assume, as before, the arrival rate and the service rate to be λ and μ, respectively.

(a) Define $u = \frac{\lambda}{\mu}$. The probability that the system is at state n (with n packets/frames) is given by $p_n = \frac{(1-u)u^n}{1-u^{k+1}}$ *if* $u < 1$

$= \frac{1}{k+1}$ *if* $u = 1$

In particular, probability that the system is idle $p_0 = \frac{1-u}{1-u^{k+1}}$

(b) Probability that a packet/frame would be discarded for lack of buffer space $p_k = \frac{(1-u)u^k}{1-u^{k+1}}$. Given this, we could obtain the effective buffer size k over a time

interval as $k_{effective} = \frac{ln\left(\frac{c-p_k}{u(1-p_k)}\right)}{ln\ u}$, $if\ u < 1$

$= \frac{1}{p_k} - 1,\ if\ u = 1$

(c) The actual arrival rate at which packets/frames enter the system $\lambda_a = (1-p_k)\lambda$

(d) The expected number of packets/frames in the system L is

$L = \frac{u[1-(k+1)u^k + ku^{k+1}]}{(1-u)(1-u^{k+1})}$ $if\ u < 1$

$= \frac{k}{2}$ $if\ u = 1$

(e) The expected queue length $L_q = L-(1-p_0)$

(f) The expected residence time $T = L/\lambda_a$

(g) The expected queuing time $T_q = L_q/\lambda_a$

(h) The utilization of the server $U = (1-p_k)u$

2.1.1.6 M/D/1 Model

In this case, for given λ, μ, ρ, we obtain the following. Note that this model results when the variance of the service time at a server is close to zero. This would be applicable to model ports in a bridge.

$$L = \rho + \frac{\rho^2}{2(1-\rho)} \qquad L_q = \frac{\rho^2}{2(1-\rho)}$$

$$T = \frac{1}{\mu} + \frac{\rho}{2\mu(1-\rho)} \qquad T_q = \frac{\rho}{2\mu(1-\rho)}$$

2.1.1.7 M/G/1 Model

For this model, assume we have obtained the variance of the service time σ_s^2 as well as the mean service time as $1/\mu$ for a server. Let $C_s^2 = \mu^2\sigma_s^2$ (for an exponential server this will equal 1). Then,

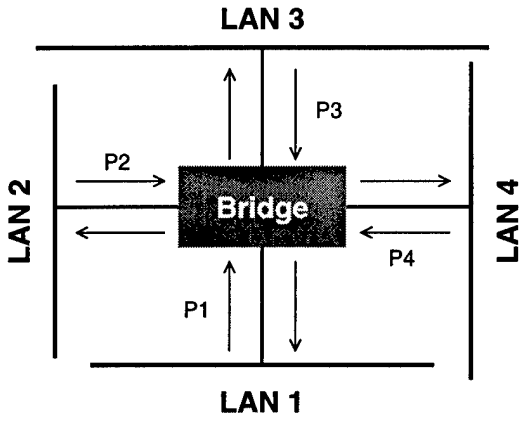
$$L = \rho + \frac{\rho^2(1+C_s^2)}{2(1-\rho)} \qquad L_q = L-\rho$$

$$T = \frac{1}{\mu} + \frac{\rho}{\mu(1-\rho)}\left(\frac{1+C_s^2}{2}\right) \qquad T_q = T - \frac{1}{\mu}$$

2.1.1.8 AN ARCHITECTURE FOR A BRIDGE

A bridge connects two or more LAN segments working entirely in the link and physical layers. In other words, there is no IP layer associated with a bridge and a bridge would receive only frames, not packets. A typical transparent bridge connecting Ethernet LANs may be viewed in Figure 2.

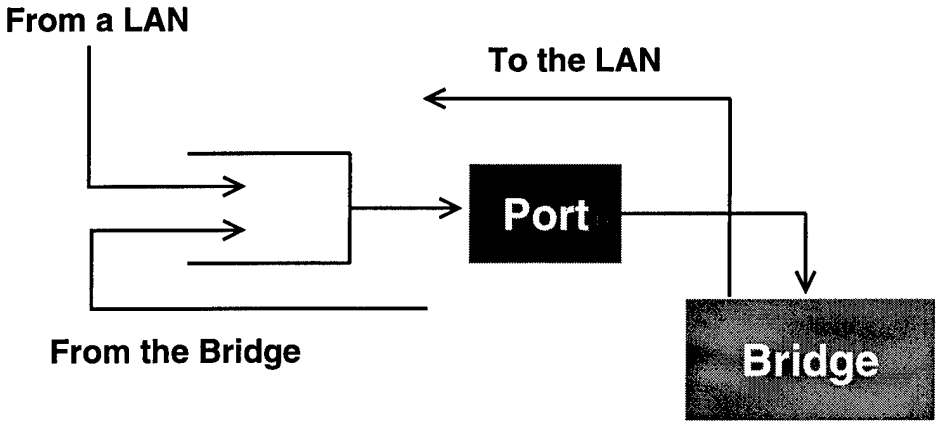
Figure 2. Typical Bridge between LANs



In the above example, a bridge with four active ports is used to connect four different LANs. For each port P_i we envisage a double tandem queue model, as for switches and nodes shown earlier, except that the service rate for the first queue would be $\mu_1 \rightarrow \infty$. This amounts to effectively discounting the first queue.

Therefore, the queue at a port P_i would be seen as a single queue. Thus each port of the bridge is equivalent to a node as modeled earlier. Diagrammatically, it appears then as a single queue (see Figure 3).

Figure 3. A Detailed View of a Port on a Bridge



2.1.2 GLOBAL PERSPECTIVE

The abstraction at the global network level appears as follows. The global network may be seen as a network belonging to a WAN (Wide Area Network) class or it may be a LAN type. Even within these two categories many different possibilities exist, each of which may qualify as a possible class for the network in question.

In this section, we will highlight the global states of networks pertaining to two common categories, the WAN architecture and the IEEE 802.3 LAN architecture. In the next report, we would address other common LAN architectural models and observe network global states in those possible architectures.

The model suite for the global network state computation would include minimally one model, which would be derived from the individual local states of the network components. We should recall that in our design the global network state *net(t)* is an appropriate abstraction based on the local states. Thus, the global state would depict mean network throughput, mean network delay, etc.

We assume the following. For the time being, we would approach the global state through a mean value analytical approach. We would assume that the arrival rate λ and the total service demand at any server would be appropriately computed. Given this, we now indicate how the relevant parameters for the global state are to be computed for our two network classes, the WAN and IEEE 802.3 LANs. As indicated earlier, the token ring architecture framework would be outlined in the next report.

2.1.2.1 WAN-CLASS NETWORKS

In WANs, the network is seen as a point-to-point, store-and-forward type, packet-switched subnet over a wide distance. Let us assume [4]:

(a) Number of reception/transmission ports of the entire system = N_{ports}

(b) Number of switches in the system = N_{switch}

(c) Total number of active nodes in the system = $N_{ports} + N_{switch}$

(d) Mean throughput rate at a node K = X_K packets/sec (reported for a local object K)

(e) System throughput rate = X packets/sec (to be derived)

(f) Mean response time (queuing time + service time delay) at a node K = $E(R_K)$ sec (reported)

(g) Mean residence time (total delay) per packet in the system = $E(R)$ sec (to be derived)

(h) Total mean external arrival rate of packets at a port K = λ_K pkts/sec (reported)

(i) Total traffic in the network system = λ packets/sec = $\sum_{K}^{N_{ports}} \lambda_K$ pkts/sec (derived)

Little's law yields the following.

$$\lambda E(R) = \sum_{K=1}^{N_{ports}} E(R_K) X_K \quad \ldots (4.1)$$

Since packets are most likely switched through more than an one internal node, we note that

$$\text{Total system throughput } X = \sum_{K=1}^{N_{switch}} X_K \geq 1 \quad \ldots (4.2)$$

with the mean delay in the network

$$E(R) = \sum_{K=1}^{N_{switch}} \frac{X_K}{\lambda} E(R_K) = \frac{X}{\lambda} \Sigma \frac{X_K}{X} E(R_K) = V \Sigma \frac{X_K}{X} E(R_K) \quad \ldots (4.3)$$

where V is the mean visit count of nodes per packet (i.e., the number of nodes visited by a generic packet). Note that the mean delay per node per packet over the system is then

$$E(R_{node}) = \Sigma \frac{X_K}{X} E(R_K) \quad \ldots (4.4)$$

This perspective is derived for the network layer. We could similarly derive the global states as seen at the physical layer where traffic flow is in bits/sec rather than pkts/sec.

Also, let us suppose D_K is the total observed service time received by a generic packet at the server K. We'll identify a threshold demand *D*. Since $XD_K = U_K \leq 1.0$ for all servers supporting a network throughput X, $D_K \leq \frac{1}{X}$ and the one with the highest value must act as a bottleneck, i.e., $D_{bottle\_neck} = \max_I \{D_I\}$ . . . (4.5). We may either report D_{bottle_neck} or any $D_K \geq D$ as potential problem parameters.

2.1.2.2 IEEE 802.3-CLASS NETWORKS

For this class of networks, we would be concerned solely with Ethernet-based LANs [5]. Instead of modeling an Ethernet via an *exact* queuing system, which would be extremely difficult and possibly not be worth the effort, we desire to capture its behavior via a queuing network whose parameters could be fine-tuned to correlate actual observation with predicted values.

Let us consider the Carrier Sense Multiple Access with Collision Detection (CSMA-CD) protocol that controls the behavior of such a network at the system level. A typical process consists of alternating contention and a frame transmission period; idle periods will not result if

we assume there is at least one busy station at all times. We would later deviate from this assumption by incorporating an appropriate factor to reflect possible idle time for the channel. Assuming a constant load for the network with an average of n active stations connected to it, we obtain the following.

The maximum probability A that one station will successfully acquire the channel is

$$A = (1 - \frac{1}{n})^{n-1} \quad \ldots (4.6a)$$

The average length of the contention interval between two successive transmissions is

$$\overline{C} = \frac{1}{A} = (1 - \frac{1}{n})^{1-n} \quad \ldots (4.6b)$$

The channel efficiency (utilization) rate on the average is

$$E(n) = \frac{F}{F + \overline{C}} = \frac{1}{1 + \frac{1}{AF}} \quad \ldots (4.6c)$$

The average delay per station is

$$\overline{D} = (F + \overline{C})(n-1)/B \quad \ldots (4.6d)$$

where B is the channel bandwidth rate. One could also post an interesting measure

$$C(n) = \frac{B}{F} E(n) \quad \ldots (4.6e)$$

This is a system-level measure, which may be of importance when we want to know how much of the maximum theoretical capacity of the network is actually used to deliver workload. Note that individual stations connected to the LAN would be modeled as shown in the Local Perspective, Section 2.1.1.

2.2 TASK 2 - OBJECT MODELING FOR ARCHITECTURE

2.2.1 MANAGED OBJECTS

For our monitor, every managed object is either a physical object like a node, or a network link. It could also be a logical object like a model of a switch or a model of a link. In fact, all possible models in which we might be interested pertain to a model class, which would be maintained within the JMAPI organization as another managed entity.

For a given network, the attributes, or current states of the managed objects (ports, switches, and links) are obtained in a virtual information store, termed the Management Information Base (MIB). In our case, we are specifically concerned with the architecture at the network and physical layers. The SNMP-brokered objects that are to be used by JMAPI are outlined below. These are the basic variables to be collected to derive models at higher levels of abstractions (for details, see RFC 1213 [7]).

Note that the following is only a subset of the objects in which we would eventually be interested. These are the objects, which are currently being used.

For Inbound Traffic from the Network to a Higher Layer

- ifInOctets (ifEntry 10) Description--Total number of octets in the interface for inbound traffic, including the framing characters.
- ifInUcastPkts (ifEntry 11) Description--Total number of sub-network unicast packets delivered to a higher level protocol.
- ifUnNUcastPkts (ifEntry 12) Description--Total number of broadcast/multicast sub-network packets delivered to a higher level protocol.
- ifInDiscards (ifEntry 13) Description--Total number of inbound packets discarded due to lack of buffer space.
- ifInErrors (ifEntry 14) Description--Total number of inbound packets discarded due to errors.
- ifInUnknownProtos (ifEntry 15) Description--Total number of inbound packets discarded due to unknown protocol specification.

At Interface

- ifSpeed (ifEntry 5) Description--The current estimate of the interface's bandwidth in bits/sec.

For Outbound Traffic

- ifOutOctets (ifEntry 16) Description--Total number of octets transmitted to the network from this interface (including framing characters).
- ifOutUcastPkts (ifEntry 17) Description--Same as for inbound traffic but now with changed direction.
- ifOutNUcastPkts (ifEntry 18) Description--Same as for inbound traffic, but now with changed direction.
- ifOutDiscards (ifEntry 19) Description—Same as for inbound traffic, but now with changed direction.

- ifOutErrors (ifEntry 20) Description—Same as for inbound traffic, but now with changed direction.

- ifOutQLen (ifEntry 21) Description--The instantaneous length of the output packet queue in packets.

For Inbound Traffic from the Network Layer to the TCP Layer

- ipINReceives (ip 3) Description--Total number of input datagrams received from interface including those in error.

- ipInHdrErrors (ip 4) Description--Total number of discarded input datagrams due to header errors.

- ipInAddrErrors (ip 5) Description--Total number of discarded input datagrams due to address errors.

- ipInUnknownProtos (ip 7) Description--Total number of discarded input datagrams due to unknown/unsupported protocols.

- ipInDiscards (ip 8) Description--Total number of discarded input datagrams due to lack of buffer space.

- ipInDelivers (ip 9) Description--Total number of input datagrams successfully delivered to IP user protocol (including ICMP).

Note that these variables pertain to input packet traffic from a node to its processes at the application level. This is distinct from the input data traffic from an interface (at the physical layer) to its link layer.

For Outbound Traffic (All Traffic from IP User to the Node for Transmission)

- ipOutRequests (ip 10) Description--Total number of IP datagrams sent to the node for transmission.

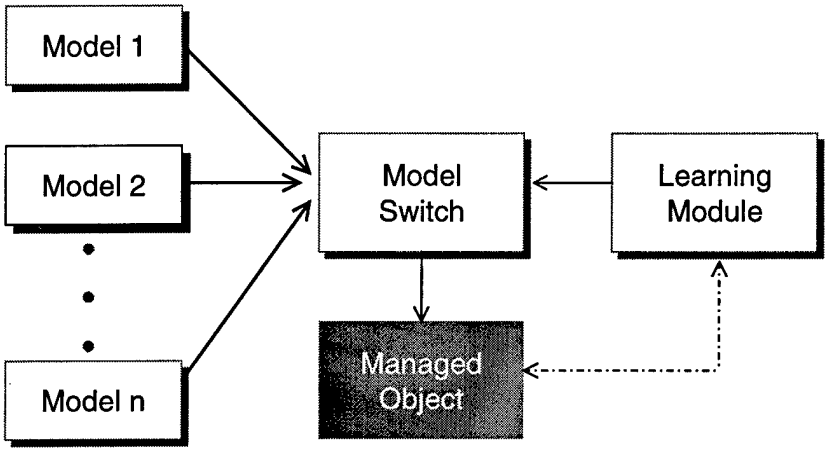
- ipOutDiscards (ip 11) Description--Total number of outbound datagram discards due to lack of buffer space.

2.2.2 MODEL ARCHITECTURE

At this stage, we noted the inadequacies of all network models currently in use. As Kleinrock [1] points out, an exact mathematical model of the system, even if we are lucky enough to develop it in the first place, may not be tractable in real situations. We always end up with an approximation even if our model is exact. Secondly, a desirable scheme may be to abandon exact mathematical models but rely on an approximate model to provide approximate but sufficiently satisfactory solutions if they correlate well with the predictions of actual

measurements [1]. In order to accomplish this, it may be necessary to learn the model parameters for an object through appropriate learning models based on the observed pasts and the performance of the learning process itself. We note that a single model to depict the behavior of a network object may not be tenable for all workload levels, for all types of traffic and for all topologies. A multimodel framework is essential. In view of these, our model architecture is proposed as follows. For a network object K, the managed object server provides a Managed Model Interface (MMI) in which a number of applicable models for that object would reside (see Figure 4). This would be true for every managed network object as well as for the model for the global state computation.

Figure 4. A View of Model Interface



The model switch would be triggered by the learning module, which would decide how to infer an object state and, given the currently inferred state, would decide which kind of model would be chosen and its parameters. Then the managed object would call that specific model object and run it with the given set of parameters for the current session. What types of models would be best suited for our network objects was investigated next.

First, we attempted to settle the issue by using the traditional Markov chain model assuming that the stochastic process $\{N(t), t \geq 0\}$ is essentially Markov (N(t) being the number of requests, packets in the system). Even though the situation improved considerably, the state-space explosion still remained the dominant issue that was difficult to ignore. Secondly, the computation hinged heavily on the distributions of the various events – the arrival or departure processes did not need to be renewal processes (independent and identically distributed), which an embedded Markov chain would minimally require. This made it difficult to proceed realistically if we needed to depart from the renewal process assumption.

Next we examined our model from another angle. We could provide a mean-value analysis (MVA) of our network environment based on the observed and estimated means. One could, on this base model, add the necessary perturbations, like fluctuations about the means, as well as ordering or scheduling constraints or correlations among different steps and different requests within the overall scheme of a typical Markovian model. It was then feasible to express the distribution functions (e.g., number of requests waiting at a node) in a product form, effectively

achieving network decomposition as if each function depended solely on the workload for that node. It was similar to expressing an arbitrary function as a Taylor series over some elemental functions. This Jackson model [2] based on the product form of distribution could be used even though one could not strictly defend it theoretically.

So we proceeded in the following manner. Keeping the overall Jackson model in view, the network would be functionally decomposed here but without embracing Jackson's assumption. We assumed that an object was essentially separate from the network, except for what it received and sent out to the network. The network is seen from the object perspective as another distinct object with which it interacts continuously but the object is, otherwise, totally independent from the other objects. The observed stochastic process embracing the object would be viewed essentially as a Markov process with its parameters appropriately determined using an intelligent learning system. Essentially, the base models would be primarily Markovian. They would be queuing-network models (M/M/1, M/G/1, and GI/G/1). Other candidate models included time-dependent stochastic Petri-net and stochastic process algebra models, along with Fractional Brownian motion-based and fractional ARIMA models.

The emphasis in Phase I of our investigation was to provide the system with a workable structure to use in selecting the most appropriate and tractable traffic model from among a collection of traffic models based on currently observed real-time data and information through collected legacy data. The model architecture as proposed was the key to our operation. This was essential for another important reason. If the operational scope for our system is to extend to the second-generation Internet it must be able to interface well with gigabit per second networks as it should with megabit per second nets. In the latter case, as pointed out in Kleinrock [3], the network is primarily bandwidth-driven whereas in gigabit regime, it is latency driven. Obviously the two require different modeling emphasis and both these distinctions should be available in the modeling suite.

2.3 TASK 3 - PROTOTYPE IMPLEMENTATION

2.3.1 INSTALLATION OF SOFTWARE

The first item of business was the installation of the software required to support the Real Time Network Management System (RTNMS). The software in question was the Java Developer's Kit version 1.1.5 (JDK), the JMAPI version 0.5 beta, Fast Forward's JDBC driver version 1.3, and SNMP agents on every machine that was to be managed.

The JDK was installed first to facilitate the installation of the other software components, all of which required the use of the Java interpreter. Then the JDBC driver was installed. This had to be installed prior to JMAPI so that the API (Application Programmer's Interface) would be able to communicate with the database server (Sybase version 11), which was already installed on the machine deemed the JMAPI server. Next came the installation of the JMAPI software itself.

Once JMAPI was installed, it was configured per the documentation's directions. One advantage was that all the software components utilized were identical to the components used in the example provided in the documentation, so configurations were almost identical.

The final requirements were the SNMP agents that allow the prototype to get information about a node. These varied from machine to machine and it was necessary to install different agents for the different platforms that were to be managed. The SNMP agents were installed on Windows95, Windows NT 4.0, and Solaris 2.5 workstations. The Windows NT machines have an SNMP agent built into the system, so it was only a matter of activating those agents. Most routers today come with an SNMP agent built in as well, including the local router and one of the AFRL routers.

2.3.2 INITIAL DATA COLLECTION

Once the software was installed, it was possible to start using the classes that JMAPI provided to collect data from any machine that was running an SNMP agent. However, this involved considerable time for the researcher to become familiar with how SNMP worked as a protocol, and to learn the JMAPI SNMP classes. The very first iteration of the data collector did little more than obtain the system description of the machine that was being queried.

During the process of implementing the data collector, a fault was discovered in how the JDK 1.1.5 worked with one of the classes within JMAPI. Between versions 1.1.4 and 1.1.5 of the JDK, the developers changed the way in which the interpreter handled Uniform Resource Locator (URL) information. The ResourceLocator class that came with JMAPI used the method found in the 1.1.4 version, which would cause NullPointerExceptions. Sun Microsystems provided an updated version of the class, which once installed, allowed the data collector to function properly.

With a working, albeit infantile data collector, it was decided to try to run the application as an applet. An attempt was made, but failed. Thinking this might have something to do with the collector, focus was turned to the demonstration applets that came as part of the JMAPI distribution. However, it was not possible to get any of these to work as an applet either. It was possible to run them as applications only. So, in the interest of continuing development, the idea of running the RTNMS from a browser was abandoned.

Work continued on expanding the collector. One major hurdle was the way SNMP stores information about interfaces. The interfaces are stored within a MIB table, and the conventional way to retrieve this information is one entry at a time, at least in SNMPv1. For purposes of this program though, all these data had to be collected simultaneously so the change in time could be observed. After extensive experimental work, the solution was found. It involved accessing all the elements at once via their entry number extension. Unfortunately, this was not documented by JMAPI and was discovered intuitively.

Modifications to the data collector continued based on analysis of the data it was collecting and a growing understanding of the SNMP variables that were being observed. This is where many misconceptions were revealed.

It was discovered that the queue length variable was the length of the queue within the interface, and that SNMP had no measurement for the IP queue, probably because there are so many potential ways to implement the IP layer. This complicated the modeling process, not only because single queue models were to be used, but also because there was now no way to get the queue information for IP.

It was also found that some of the SNMP variables collected were unnecessary while some that were initially overlooked had to be retrieved. The number of routing table entries that are removed from the routing table was deemed unimportant; it was first thought that this variable measured some form of packet dropping. It had to do with the source and destination routes instead. All of the possible errors that occurred within the interface and IP layers were not being collected. These errors were needed for more accurate models; the number of discarded packets due to buffer overflows was not sufficient.

It was also at this stage that a rudimentary prediction algorithm was implemented to test its accuracy. Samples showed that the algorithm's performance varied based on the type of network activity. Adjusting the weight of the previously predicted and actual readings showed it was possible to get better results with a variable weight. So the average time error over the set of the collected data was used to determine how to adjust the weight automatically. This scheme was used in the final prototype.

2.3.3 CREATION OF JMAPI-MANAGED OBJECTS

With the data collection proceeding, focus was shifted toward how the data were to be stored. This is where JMAPI's ability to communicate with a database came into play. Actually, JMAPI requires that it be tied into a database before it will run properly and it stores data in the database via the ManagedObject class. JMAPI comes with an entire hierarchy based on ManagedObject. However, this hierarchy was being completely rewritten at Sun and they advised subclassing ManagedObject directly.

Documentation for creating managed objects was also vague due to JMAPI still being in its infancy. The hard copy documentation, which was more of a tutorial-type document, gave an initial starting point, but was outdated and much of the code illustrated was deprecated. Fortunately, the online reference had the current class definitions and method calls. By doing a little experimentation, it was possible to create a functional managed object. This first managed object was little more than an integer value, but demonstrated that it was possible to "set and get" information from the JMAPI database.

While importing the managed object into the database, it was discovered that the JDBC driver installed was inadequate. The JDBC driver used was the company's shareware version and could only handle two simultaneous connections to the database. Consulting with Sun again, it

was learned that JMAPI requires 5 or 6 connections when importing a managed object. The JDBC driver company was contacted and they sent a 30-day, 50-connection version of the driver so testing could be done to insure this was the problem with importing managed objects. The new driver was installed and the managed object was successfully imported. This led to the purchase of Fast Forward's 15-connection driver.

A large amount of traffic was observed on the loopback port on the JMAPI server during Phase I prototype execution. After researching the problem, it was found that the JDBC driver was creating this extra traffic. FastForward, a Type IV driver from Connect Software, is the JDBC recommended by Sun when using JMAPI. It provides direct access to Sybase SQL (Standard Query Language) servers. FastForward works by directly transferring and receiving information from Java to the SQL Server using TCP/IP sockets. The format of data passed back and forth is Tabular Data Stream format (TDS) [6]. This may be a reason why access to the database was less than sufficient. Therefore we may look into other JDBC drivers for a more efficient way of accessing data in Phase II.

Once it was shown that the database was working properly and managed objects could be stored there, design of the managed objects ensued. A rough draft of all necessary managed objects for the network management system was conceived. At this point more refinement of the mathematical models and the way SNMP represents the interface and IP layers was required.

One problem was the fact that a node may have multiple interfaces, each of these having its own queue. Significant work had to be done to adapt the single queue models to the now multiqueued node. A queue length also had to be mathematically derived for the IP layer.

The IP layer has the potential to fragment and/or reassemble packets before passing them down to the interface or up to higher layer protocols. This affected comparisons of arrival and service rate because the rates were in terms of different types of packets. So a fresh look at how the IP layer fragments and reassembles packets was required. A multiplier was added to the model to compensate for the discrepancy between the number of packets into and out of the IP and interfaces. The multiplier going from the interfaces to the IP layer is less than one. The multiplier from IP to the interfaces is between one and two. This approximates the number of packets that are resent by the datalink layer, for which there is no SNMP data.

Another multiplier had to be used to calculate the percentage of packets going to each of the interfaces from the IP layer. This was used in conjunction with the datalink multiplier to determine how much of the IP output was going to each interface. This was required because there was no SNMP variable to indicate the number of packets to a particular interface.

The complexity of how the IP layer functions was particularly difficult to model. With IP both fragmenting and reassembling packets, some of the details of how IP handles packets gets lost. In addition, IP does not maintain a count of the packets that are lost because of reassembly errors. So attempts to extrapolate this information were made. In addition to this, some of the machines being monitored represented the reassembly information differently than the RFC 1213 [7] specifies. The Sun workstation does not give the number of fragments that need to be reassembled; instead it gives the number of reassembled packets. It is unknown why Sun would

choose to do this in their MIB; perhaps it has something to do with their implementation of the IP layer.

2.3.4 INTEGRATION OF COMPONENTS

The managed objects were implemented and loaded into the database. These objects were created the same way normal tables in a database would be created, each one having its own primary and foreign keys. JMAPI allows the use of associations and relationships between managed objects, but with the documentation being sparse, it was deemed easier to create the objects with keys and allow application software to ensure integrity constraints.

With all the necessary managed objects created within the database, it was time to implement the code to do all the work.

The initial collector would read in the user's information on a node, (name, IP address, node type, and read community). It would then use that information to query the node to get the number of interfaces the node has, the IP addresses of all those nodes, speeds, etc., and to populate the appropriate managed objects with that information.

With this information, whole topologies were created, and the user could select which topology he/she wanted to monitor. In order to monitor each node as efficiently as possible, the monitoring software was written to create a thread for each node that was being monitored. Each thread is responsible for polling the SNMP agent of the node it is monitoring as well as sending the received data to the JMAPI server to be stored.

Even though the process of collecting data from the SNMP agents is fairly efficient, the process of using JMAPI to store and retrieve such an enormous amount of data is not. Even with only six nodes, the time it took to collect and store the data was in the order of minutes. Most of this time was due to the complex process of using the JMAPI calls to perform the "sets and gets."

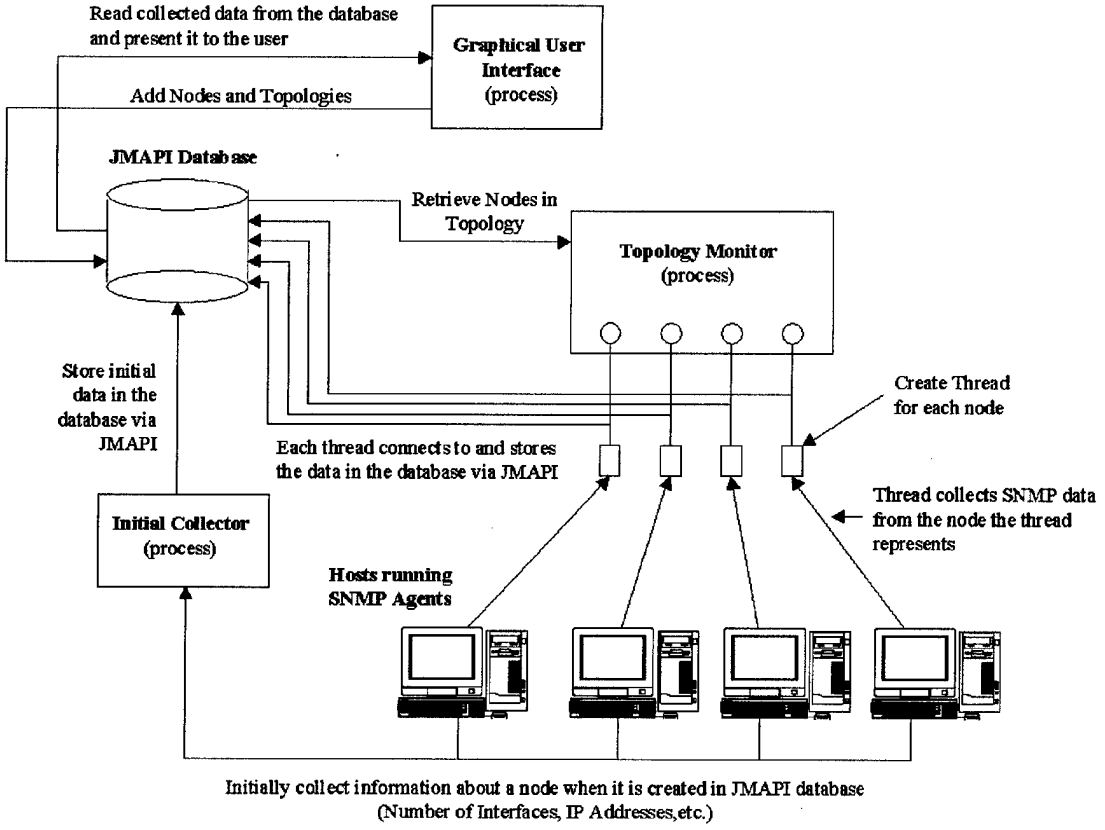
A network administrator could then look at all this information with the implementation of the graphical user interface (GUI). The GUI uses the Java Abstract Windowing Toolkit (AWT) package as well as the PropertyBook classes that come with JMAPI. This interface provides an efficient and intuitive way for an administrator to access and analyze all the information that is being collected about the network.

2.3.5 EXPLANATION OF IMPLEMENTATION

The software used included the JDK version 1.1.5 and the JMAPI version 0.5, both from Sun. In addition, there was Fast Forward's JDBC version 1.3 allowing JMAPI to communicate with the Sybase database version 11. Sybase is used to store all the information about the machines being monitored on the network. SNMP agents were installed on multiple platforms so that the information about that machine could be collected.

JMAPI is a tool created to aid in the development of network resource and management applications. It simplifies this task by abstracting every aspect of a network into something known as a managed object. Anything related to the network is a subclass of this managed object. Doing this allows the developer to focus on the intricacies of the network to be managed and not how the information will be stored in the database; JMAPI handles all these details.

Figure 5. Explanation of Implementation



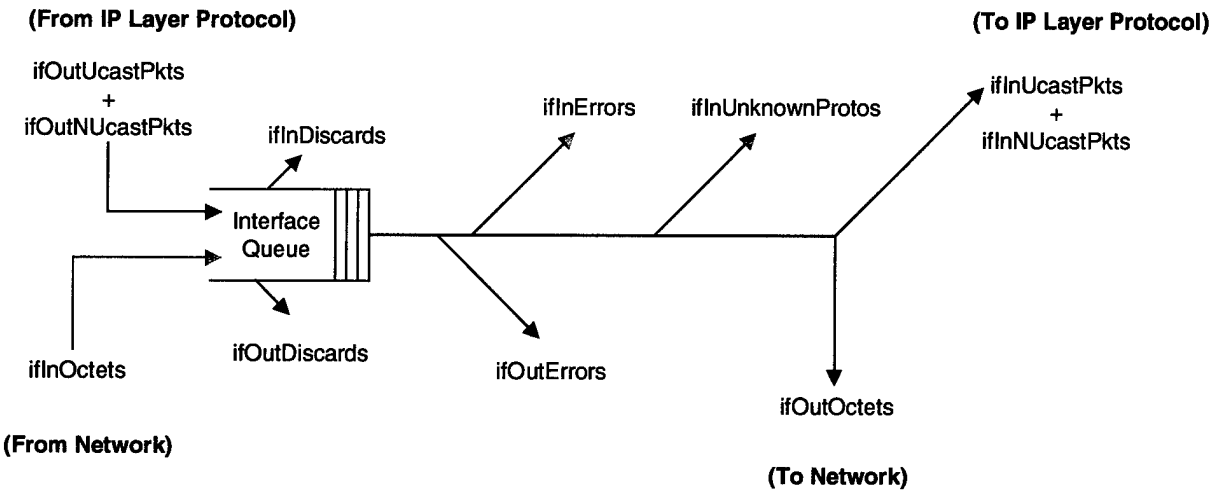
The application components sit on top of the JMAPI interface. The Initial Collector Process uses the SNMP classes that came with JMAPI to collect information on a host that is currently being added to the database. Once a node has been added to the database, it can be added to one or more topologies and monitored. The initial collector can also be run from within the GUI.

The Topology Monitor is the main part of the Management System. It will take the topology the user wishes to monitor and will run off a thread for each of the nodes that are within that topology. Each thread will communicate with the SNMP agent on the host the thread is monitoring. The thread collects the necessary data and then communicates to JMAPI in order to get the information stored persistently.

The GUI can then access the data via JMAPI. The interface supports the ability to graph time series of the collected data, as well as display the network and its utilization based on the data.

The Interface Layer is an abstraction of how the interface card within a computer handles incoming and outgoing data. Data come into an interface from the network and from higher layer protocols within the protocol stack. Data from the network are in the form of octets; data from above are in the form of packets (octets are the number of bytes). Packets are groups of bytes representing some sort of information. For SNMP version 1.0, the packets can be Unicast or Non-Unicast. For purposes of the RTNMS, these are summed together. Focus is on amount of traffic, not types of traffic.

Figure 6. Interfaces According to SNMP Variables



As stated before, traffic from the network is in terms of octets. A stream of octets makes up a packet. This stream is stored in the interface's buffer until it can be processed by the interface. If too many streams come into an interface and the buffer is not large enough to accommodate them, some of the streams (packets) will be discarded. The SNMP variable ifInDiscards keeps a count of the number of packets that the interface loses in this manner. When the interface is able to process a packet in the queue, it checks the packet for errors and whether or not the packet is in an understandable protocol. SNMP variables ifInErrors and ifInUnknownProtos count the packets lost when they do not pass these tests. If the packet makes it past this point, it is sent to the next protocol up on the protocol stack. The packets that make it are counted in the ifInUcastPkts and ifInNUcastPkts SNMP variables, depending on which type of packet it is.

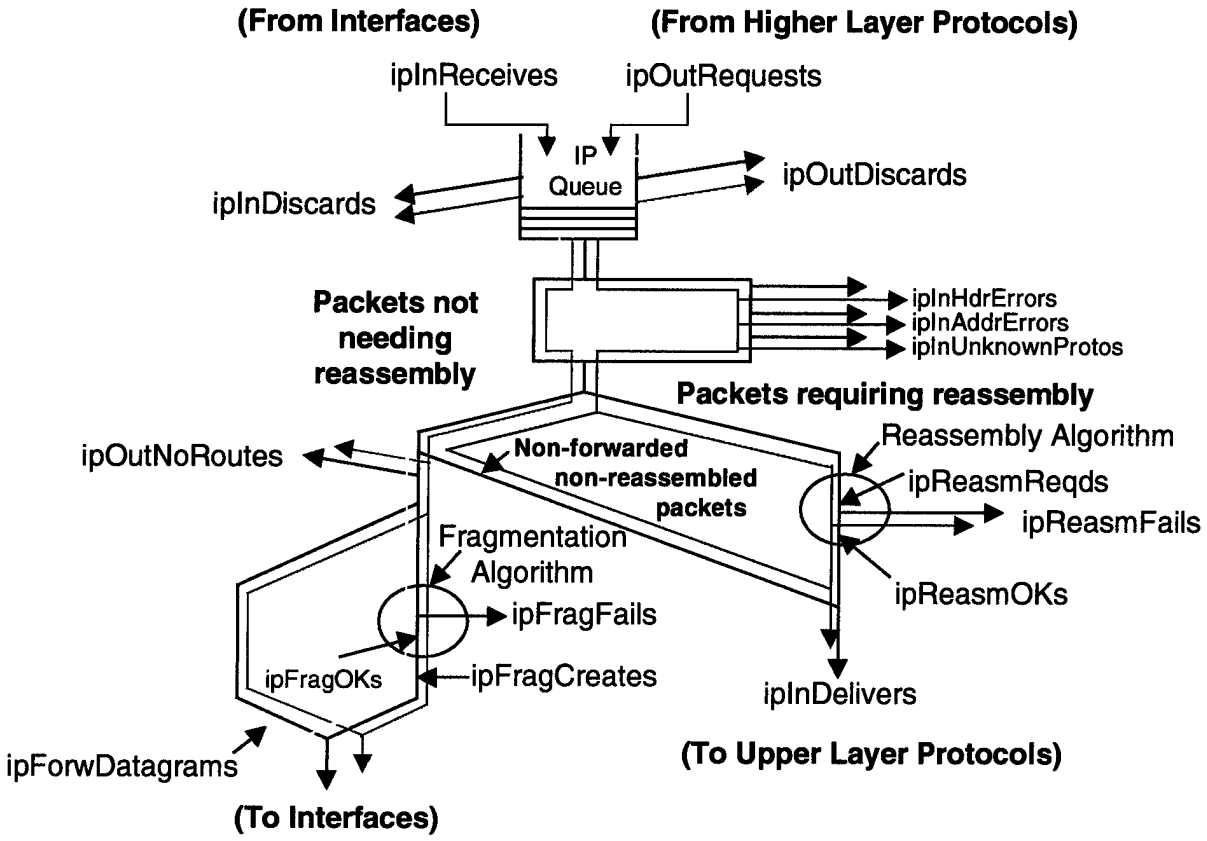
When a packet comes into the interface from the higher protocol, it is counted in either the ifOutUcastPkts or ifOutNUcastPkts, again depending on the type of packet it is. It gets stored in the same buffer in which the input from the network is stored until the interface can process it. If the buffer overflows, and some of these outbound packets are lost, they are counted in the ifOutDiscards variable. If the interface is processing the outbound packet and finds some error in the packet, it will discard the packet and increment the ifOutErrors variable. Finally, all the octets that make up the packets that do get processed are counted in the ifOutOctets variable.

Since it was necessary to know the number of packets travelling between the interface and the network, they had to be derived. The number of packets coming into the interface from the

network can be calculated by summing the number of packets going to higher protocols (ifInUcastPkts and ifOutUcastPkts), ifInUnknownProtos, ifInErrors, and ifInDiscards. Adding all the good packets to all the dropped packets should give a count of the total number of packets coming into the interface from the network.

To calculate the packets that actually make it out to the network from the interface, the number of packets discarded (ifOutDiscards) and dropped because of errors in the packet (ifOutErrors) were subtracted from all the packets coming into the interface from the higher protocol (ifOutUcastPkts and ifOutNUcastPkts). Subtracting the bad packets from the total number of packets should give the number of packets actually going out to the network.

Figure 7. The IP Layer According to SNMP Variables



The IP layer is much more complicated. Depending on which task the machine is performing, the IP layer may perform differently. If the machine is a router, a great deal of packet forwarding will occur. It the machine is a host, then no packet forwarding is allowed to occur. However, if the host is functioning as a gateway, then it will forward packets. This complex behavior, in addition to the vagueness of some of the SNMP variables, led to significant confusion.

Figure 7 shows the different directions packets can take through the IP layer. The ipInReceives SNMP variable counts all the packets coming into the IP layer from the interfaces. This includes

packets from all of the machine's interfaces. These packets can either be destined for the higher protocols of the machine or forwarded out to the network. The blue line on the graph represents the path packets would take through the IP if they were meant for higher protocols. The yellow line is the path forwarded packets would traverse. For hosts, the yellow path would be nonexistent. For routers, the blue path would see very little traffic, though some would be present. SNMP requests would show up as such. For a gateway, traffic would be possible on both routes.

Packets coming from the higher level protocols enter the IP layer and are counted via the ipOutRequests variable. This is shown on the diagram as the green path.

The crux of the problem is that the IP layer can fragment packets being sent to the interfaces into smaller packets and reassemble packets that are going up to the higher layers into larger packets. This is all done because some networks cannot handle larger packet sizes and it is the IP layer's responsibility to break those packets down so they can be transmitted over those less capable networks. This also means that the packet counters at different points of the IP layer are not counting packets the same way. For example, ipInReceives counts the smaller packets, but ipInDelivers counts the same packets after they have been reassembled. As an example a quantity of little packets come into IP and ipInReceives counts them. Assume that no discards or errors occur, and some of the packets are reassembled. ipInDelivers will show a smaller number of packets going to upper layer protocols than ipInReceives said came into IP, even though there were no losses. This is because some of the smaller packets are consolidated into their larger, original size. So ipInDelivers is counting the fewer, larger packets, while ipInReceives is counting the smaller, more numerous packets.

For the mathematical models to work properly, the inputs and outputs of the IP layer must be in the same type of packets (small or large). Since the smaller packets correspond to the packets the interfaces send to and receive of the IP layer, it was decided to try to convert all of the inputs into the smaller packet size. For a host, this was not much of a problem, but for routers and gateways, the task was much more difficult.

When dealing with a host, the yellow path in Figure 7 can be eliminated because hosts are not allowed to forward packets. The ipOutRequests could be calculated in terms of small packets using the SNMP variables like this.

$$\textit{Little Packets from Higher Protocols} = \textit{ipOutRequests} + (\textit{ipFragCreates} - \textit{ipFragOKs})$$

This formula takes ipFragCreates, which is the total number of packets created by the fragmentation algorithm and subtracts ipFragOKs, which is the number of packets that were successfully fragmented. This results in the number of additional packets created due to fragmentation. When added to ipOutRequests, it gives the number of little packets coming from the higher protocols.

SNMP does not have a variable to determine how many packets are being sent to the interfaces from IP, so this number is derived as well. To get this number, the SNMP variables are used thusly.

Little Packets to the Interfaces = ipOutRequests + (ipFragCreates – ipFragOKs) – ipOutNoRoutes – ipOutDiscards

The first part of the formula is the number of little packets from the higher protocols. Subtracting ipOutNoRoutes and ipOutDiscards gives the number of small packets that make it through IP and can be sent to the interfaces. These formulas only work for machines that are running as hosts.

Packets coming into the IP layer from the interfaces are counted in terms of little packets already, so no derivation of small packets is required.

Packets going from IP to the higher layer protocols are measured in terms of reassembled (big) packets. This number can be retrieved from the ipInDelivers variable. Getting this number in terms of small packets is problematic. This is because the reassembly algorithm cannot count the packets that could not be reassembled. Therefore, SNMP can only count the number of times the reassembly algorithm incurs a failure. These failures in no way reflect the number packets that are lost due to those failures. The best way to illustrate why this is the case is to outline how the reassembly algorithm works.

When the reassembly algorithm receives a packet fragment that needs to be reassembled, it creates a queue and sticks that packet into it. Packet fragments are, in their own right, packets, only smaller. It will then place additional fragments belonging to that packet into the queue. When all of the fragments have been collected, the algorithm will assemble the packet and pass it along to the next higher protocol. The algorithm will only hold the packet fragments in the queue for a finite amount of time however. If this time limit is exceeded, the algorithm will drop all the fragments in the queue and increment the ipReasmFails variable. The problem arises when failures occur. There is no way to determine how many fragments were lost because of errors or how many fragments made it through to the reassembly algorithm to become the larger packets. For the purposes of the prototype, it was assumed that no packets needed reassembly. Under this assumption, ipInDelivers would therefore be in terms of packet fragments. The data indicated that reassembly was indeed a rare occurrence for the network we were monitoring. It is obvious that more work is needed on this aspect of the IP layer.

Another problem was encountered in the way some of the monitored nodes represented one of SNMP variables. In most cases, we know the number of fragments that need reassembly; the count is stored in the ipReasmReqds variable. Observation of the collected data shows that ipReasmReqds on the Sun workstations is a count of the number of whole packets that need reassembly, not the number of packet fragments that need reassembly.

The number of packet fragments for the Sun machines can be derived, as long as the machine is functioning as a host. To get the number of fragments, the following formulas were used.

totPktIn = ipInReceives – ipInDiscards – ipInHdrErrors – ipInAddrErrors – ipInUnknownProtos

Packets needing reassembly = totPktIn – ipInDelivers + ipReasmOKs

If the machine is operating as a gateway, then there is no way to ascertain the number of packet fragments needing reassembly, because it is not possible to determine how many of the ipInReceives packets are going to higher protocols and how many are being forwarded.

While considerable work was done, particularly at the IP layer, to model the way nodes behave in terms of SNMP variables, it is evident that more work is needed. More definitive methods for calculating packet fragments going into and coming out of the IP layer are required. Another possibility might be to look into version 2 of SNMP or to implement a customized agent to retrieve the information required for the mathematical models.

3.0 REFERENCES

1. L. Kleinrock, "On the Modeling and Analysis of Computer Networks," *Proc. IEEE*, pp 1179-1190, 1993.

2. Allen, *Probability, Statistics and Queuing theory*. Academic Press 1978.

3. L. Kleinrock, "Channel Efficiency for LANs," *Local Area & Multiple Access Networks*, R.L. Pickholtz, Ed. Rockville, MD: Computer Science Press, 1986, pp 31-41.

4. E. D. Lazowska, J. Zahorjan, G. Scott Graham and K. C. Sevcik, *Quantitative Systems Performance*, Prentice-Hall 1984.

5. S. Tanenbaum, *Computer Networks,* 3rd Edn., Prentice-Hall 1996.

6. Sood, Mukul, "JDBC Drivers and Web Security," *Dr Dobb's Journal,* July 1998, pp 90-95.

7. RFC 1213, http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1213.txt, "Management Information Base for Network Management of TCP/IP-based Internets: MIBII."

4.0 GLOSSARY

API	Application Programmer's Interface
AWT	Abstract Windowing Toolkit
CDRL	Contract Data Requirements List
CSMA-CD	Carrier Sense Multiple Access with Collision Detection
FIFO	First In, First Out
GUI	Graphical User Interface
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IP	Interface Protocol
JDBC	Java Database Connectivity
JDK	Java Developer's Toolkit
JMAPI	Java Management Application Programmer's Interface
LAN	Local Area Network
MIB	Management Information Base
MMI	Managed Model Interface
MVA	Mean-value Analysis
RTNMS	Real-Time Network Management System
SAP	Service Access Point
SNMP	Simple Network Management Protocol
SQL	Standard Query Language
SUNY	State University of New York
TCP	Transmission Control Protocol
TDS	Tabular Data Stream
URL	Uniform Resource Locator
WAN	Wide Area Network